Weeks 8: Design Patterns Examples

Patterns to be discussed today:

- **Structural:**
  - Facade
  - Proxy
  - Bridge

The slides are adapted from the book titled: *The Design Patterns (Java Companion)* by JAMES W. COOPER ([http://www.patterndepot.com/put/8/JavaPatterns.htm](http://www.patterndepot.com/put/8/JavaPatterns.htm))

UML Models for patterns are borrowed from the book titled: *Design Patterns - Elements of Reusable Object-Oriented Software* by Gamma et al
The Facade Pattern

• **Usage:** The Facade pattern helps reduce complexity by providing a simplified interface to the subsystems. This simplification may in some cases reduce the flexibility of the underlying classes, but usually provides all the function needed for all but the most sophisticated users.
Motivation for Facade

- The classes in the java.sql package provide an excellent example of a set of quite low level classes that interact in a convoluted manner, as shown below:
1) To connect to a database, you use an instance of Connection.
2) Then, to find out the names of the database tables and fields, you need to get an instance of the DatabaseMetadata class from the Connection.
3) Next, to issue a query, you compose the SQL query string and use the Connection to create a Statement class.
4) By executing the statement, you obtain a ResultSet class.
5) To find out the names of the column rows in that ResultSet, you need to obtain an instance of the ResultsetMetadata class.

➔ It can be difficult to juggle all these classes!
➔ Also, since most method calls throw exceptions, the coding can become messy.
However, by designing a Facade consisting of a Database class and a resultSet class (note the lowercase “r”), we can build a much more usable system.
Motivation for Facade (continued)

• Let’s consider how we connect to a database. We first must load the database driver:

```java
try{Class.forName(driver);}  //load the Bridge driver
catch (Exception e)
{System.out.println(e.getMessage());}
```

and then use the Connection class to connect to a database. We also obtain the database metadata to find out more about the database:

```java
try {con = DriverManager.getConnection(url);
dma =con.getMetaData();  //get the meta data
}
catch (Exception e)
{System.out.println(e.getMessage());}
```
Motivation for Facade (continued)

• If we want to list the names of the tables in the database, we need to call the `getTables` method on the database metadata class,
  ▪ This returns a ResultSet object.
• Finally, to get the list of names we have to iterate through that object, making sure that we obtain only user table names, and exclude internal system tables.

Vector tname = new Vector();
try {
    results = new ResultSet(dma.getTables(catalog, null, "\%", types));
} catch (Exception e) {System.out.println(e);}
while (results.hasMoreElements())
    tname.addElement(
        results.getColumnValue("TABLE_NAME"));
Example for Facade

• As seen from the example, the code becomes quite complex to manage, and we haven’t even issued any queries yet.
• A simplifying assumption is that the exceptions these database class methods throw do not need complex handling.
  ▪ For the most part, the methods will work without error unless the network connection to the database fails.
  ▷ Thus, we can wrap all of these methods in classes in which we simply print out the infrequent errors and take no further action.

This makes it possible to write two simple enclosing classes containing all significant methods of the Connection, ResultSet, Statement and Metadata classes.
Example for Facade (continued)

• The Database class:

```java
Class Database {
    public Database(String driver) //constructor
    public void Open(String url, String cat);
    public String[] getTableNames();
    public String[] getColumnNames(String table);
    public String getColumnName(String table,
                                  String columnName);
    public String getNextValue(String columnName);
    public resultSet Execute(String sql);
}
```

• The resultSet class:

```java
class resultSet {
    public resultSet(ResultSet rset) //constructor
    public String[] getMetaData();
    public boolean hasMoreElements();
    public String[] nextElement();
    public String getColumnValue(String columnName);
    public String getColumnValue(int i);
}
```
The Proxy Pattern

- **Usage:** to represent a complex object by a simpler one.

- A Proxy usually has the same methods as the object it represents, and once the object is loaded, it passes on the method calls from the Proxy to the actual object.

- There are several cases where a Proxy can be useful:
  - If an object, such as a large image, takes a long time to load.
  - If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
  - If the object has limited access rights, the proxy can validate the access permissions for that user.
Structure

Possible object diagram:
Example for Proxy

• We want to write a program to display an image on a JPanel when it is loaded. Rather than loading the image directly, we use a class called ImageProxy to defer loading and instead draw a rectangle around the image area until loading is completed.

```java
public class ProxyDisplay extends JFrame {
    public ProxyDisplay() {
        super("Display proxied image");
        JPanel p = new JPanel();
        getContentPane().add(p);
        p.setLayout(new BorderLayout());
        ImageProxy image = new ImageProxy(this, "elliott.jpg", 321, 271);
        p.add("Center", image);
        setSize(400, 400);
        setVisible(true);
    }
```
Example for Proxy (continued)

- We create an instance of the ImageProxy just as we would have for an Image, and add it to the enclosing JPanel exactly like an actual image.
- The ImageProxy class sets up image loading and creates a MediaTracker object to follow the loading process:

```java
public ImageProxy(JFrame f, String filename, int w, int h)
{
    height = h;
    width = w;
    frame = f;

    tracker = new MediaTracker(f);
    img = Toolkit.getDefaultToolkit().getImage(filename);
    tracker.addImage(img, 0); //watch for image loading

    imageCheck = new Thread(this);
    imageCheck.start(); //start 2nd thread monitor

    //this begins actual image loading
    try{
        tracker.waitForID(0,1);
    }
    catch(InterruptedException e){}
}
```
Example for Proxy (continued)

• The \textit{waitForID} method of the MediaTracker actually initiates loading.

• We put in a minimum wait time of 1 msec so that we can minimize apparent program delays.

• The constructor also creates a separate thread \textit{imageCheck} that checks the loading status every few milliseconds, and starts that thread running.

• For the purposes of this illustration, we may elect to slow the polling time down to 1 second so that we can see the program draw the rectangle and then refresh the final image.
Example for Proxy (continued)

```java
public void run()
{
    //this thread monitors image loading
    //and repaints when the image is done
    try{
        Thread.sleep(1000);
        while(! tracker.checkID(0))
            Thread.sleep(1000);
    }
    catch(Exception e){}
    repaint();
}
```

- Finally, the Proxy is derived from the JPanel component, and therefore, naturally has a `paint` method. In this method, we draw a rectangle if the image is not loaded. If the image has been loaded, we erase the rectangle and draw the image instead (next slide ...
Example for Proxy (continued)

```java
public void paint(Graphics g)
{
    if (tracker.checkID(0))
    {
        height = img.getHeight(frame);  //get height
        width = img.getWidth(frame);    //and width
        g.setColor(Color.lightGray);    //erase box
        g.fillRect(0, 0, width, height);
        g.drawImage(img, 0, 0, frame);    //draw image
    }
    else
    {
        //draw box outlining image if not loaded yet
        g.drawRect(0, 0, width-1, height-1);
    }
}
```
Example for Proxy (continued)

The program’s two states are illustrated below.
The Bridge Pattern

- **Usage:** to separate the interface of class from its implementation, so that either can be varied separately.

- Not to be confused with the Adapter pattern: the two patterns are similar in that a class is used to convert one kind of interface to another.

- **However,** the intent of the adapter pattern is to make one or more classes’ interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class’s interface from its implementation.
Structure
Example for Bridge

• Suppose we have a program that displays some information about a set of products. We need two kinds of displays from our product data, a customer view that is just the list of products, and an executive view which also shows the number of units shipped.

<table>
<thead>
<tr>
<th>Customer view</th>
<th>Executive view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brass plated widgets</td>
<td>Brass plated w...</td>
</tr>
<tr>
<td>Furled frammis</td>
<td>Furled framm...</td>
</tr>
<tr>
<td>Detailed rat brushes</td>
<td>Detailed rat br...</td>
</tr>
<tr>
<td>Zero-based hex dumps</td>
<td>Zero-based he...</td>
</tr>
<tr>
<td>Anterior antelope collars</td>
<td>Anterior antelo...</td>
</tr>
<tr>
<td>Washable softwear</td>
<td>Washable soft...</td>
</tr>
<tr>
<td>Steel-toed wing-tips</td>
<td>Steel-toed win...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brass plated widgets</td>
<td>1,000,076</td>
</tr>
<tr>
<td>Furled frammis</td>
<td>75,000</td>
</tr>
<tr>
<td>Detailed rat brushes</td>
<td>700</td>
</tr>
<tr>
<td>Zero-based hex dumps</td>
<td>80,000</td>
</tr>
<tr>
<td>Anterior antelope collars</td>
<td>578</td>
</tr>
<tr>
<td>Washable softwear</td>
<td>789,000</td>
</tr>
<tr>
<td>Steel-toed wing-tips</td>
<td>456,886</td>
</tr>
</tbody>
</table>
Example for Bridge (continued)

• Create instances of a list and a table from classes derived from JList and JTable but designed to parse apart the names and the quantities of data.

pleft.setLayout(new BorderLayout());
pright.setLayout(new BorderLayout());

//add in customer view as list box
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center", new productList(prod));

//add in execute view as table
pright.add("North", new JLabel("Executive view"));
pright.add("Center", new productTable(prod));
Example for Bridge (continued)

- Derive the `productList` class directly from the `JawtList` class we saw last week, so that the Vector containing the list of products is the only input to the class.

```java
class productList extends JawtList
{
    public productList(Vector products)
    {
        super(products.size());  // for compatibility
        for (int i = 0; i < products.size(); i++)
        {
            // take each string apart and keep only
            // the product names, discarding the quantities
            String s = (String)products.elementAt(i);
            // separate qty from name
            int index = s.indexOf("--");
            if(index > 0)
            {
                add(s.substring(0, index));
            }
            else
            {
                add(s);
            }
        }
    }
}
```
Example for Bridge (continued)

• Suppose we need to make some changes in the way the data is displayed
  ➔ e.g., want to have the products displayed in alphabetical order.
  ➔ Need to either modify or subclass both of our classes.
    ➔ This can quickly become a maintenance headache, esp. if more than two such displays are eventually needed.

• Rather than deriving new classes whenever a change needs to be made the displays, we build a single bridge to handle this.
Example for Bridge (continued)

- Want the bridge class to return an appropriate visual component. we’ll use the scroll pane class:
  
  ```java
  public class listBridge extends JScrollPane
  ```

- When designing a bridge, we have to decide how the bridge will determine which of the several classes it is to instantiate. Could decide based on the values/quantities of data to be displayed, or some simple constants. Here we define two constants inside the listBridge class:
  
  ```java
  static public final int TABLE = 1, LIST = 2;
  ```
Example for Bridge (continued)

- We keep the main program constructor much the same, replacing specialized classes with two calls to the constructor of our new listBridge class:

  ```java
  pleft.add("North", new JLabel("Customer view"));
  pleft.add("Center",
          new listBridge(prod, listBridge.LIST));

  //add in execute view as table
  pright.add("North", new JLabel("Executive view"));
  pright.add("Center",
          new listBridge(prod, listBridge.TABLE));
  ```

- The constructor for listBridge class is then:

  ```java
  public listBridge(Vector v, int table_type) {
      Vector sort = sortVector(v); //sort the vector
      if (table_type == LIST)
          getViewport().add(makeList(sort)); //make list
      if (table_type == TABLE)
          getViewport().add(makeTable(sort)); //make table
  }
  ```